CSCI 132 Data Structures—Lab #4

Introduction

In today's lab you will practice working with linked stack and array-based queue classes.

Setup:

- Log onto Logos, and copy the code for the lab:
  ```
  cp -r ~csci132/labs/lab4 ~/labs/
  cd ~/labs/lab4/
  ```

Working with a linked Stack

For the first task, we will use a `LinkedStack` class to represent *stack* with a linked list instead of an array. Figure 1 gives a high level picture of a Linked Stack.

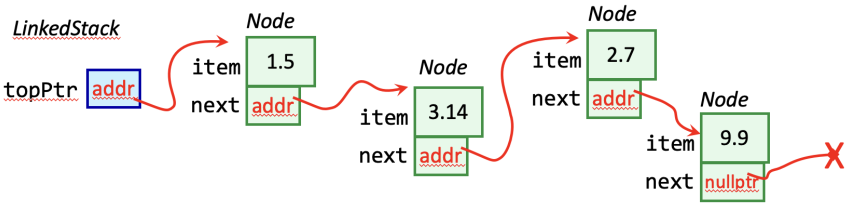

Figure 1: Implementation of a Linked Stack

Checkpoint 1:
Add a new `const` method to the `LinkedStack` class named `isEqual()` that takes a parameter of type `const LinkedStack<ItemType>&` and returns true if the given stack has the same the entries in the same order as the current stack (i.e., `this`).

Keep in mind the following:

- The necessary prototype has been added to `LinkedStack.h`, and the method is given as a stub in `LinkedStack.cpp`.
- The method is `const`, so your code should not modify the current stack's member variables. The parameter should also be `const`, so your should not modify the stack passed as a parameter either.
- To implement the `isEqual()` function, create two `Node` pointers, one for each linked list, and use a single loop that moves both pointers along the two stacks' linked lists, comparing each of the entries.

You can compile and run a test program using:

```
clang++ -g -Wall StackTest.cpp -o stacktest && ./stacktest
```

Note: This single command compiles then runs the program. In bash, `commandA && commandB` means, "run `commandA`, AND if that succeeds, then run `commandB`". You could also just use two terminal commands, on separate lines, as usual.

Sample Output

Your output should look as follows:

```
Contents of Stack 1: 0 2 4 6 8 10 12 14 16 18 20
Contents of Stack 2: 0 2 4 6 8 10 12 14 16 18 20
Contents of Stack 3: 1 3 5 7 9 11 13 15 17 19


Testing isEqual for Stack 1 and Stack 2.
Correctly returns true.

Testing isEqual for Stack 1 and Stack 3.
Correctly returns false.

Adding 100 to Stack 2.
Testing isEqual for Stack 1 and (updated) Stack 2.
Incorrectly returns true.

Testing isEqual for Stack 1 and an empty stack.
Correctly returns false.

Testing isEqual for two empty stacks.
Correctly returns true.
```

Prime Factors: A Stack client

`Primes.cpp` presents starting code for a program that uses a stack to read in an integer and print out all of its prime divisors in descending order.

Checkpoint 2:
Complete this implementation by filling out code in `Primes.cpp`.

Keep in mind the following:

- The smallest divisor greater than 1 of any integer is guaranteed to be a prime. You can check if a factor evenly divides a number using the MOD operator (%, aka "remainder").

  For example, since 10 *is divisible by* 2*,* $10\%2 = 0$*.*

  But since 11 *is not divisible by* 2 *and gives a remainder of* 1 *upon division,* $11\%2 = 1$*.*

- Once a factor is found, you need only find the divisors of your number/factor.

- A given factor may divide the number more than once.

- When a factor is found, push it onto the stack

- After all factors have been found, print them in reverse order by accessing the stack.

  (Note: You will empty the stack in the process of printing out the values).

You have been given several variables to start you off:

- `number` is used to read in the number to be factored. The value of **number** should not change during the program.
- `factor` should be used to hold the value of each potential factor as it is tested.
- `result` should be used to hold the current number to be tested for the next prime factor, after division by **factor**, i.e., **result = result/factor.**
- `primeStack` is a stack to store each prime factor as it is found. Make sure to check that the operation was successful when you push a factor onto the stack; if the stack operation fails, stop the search early and print the (partial) results.

Compile and run your program using: `clang++ -g -Wall Primes.cpp -o prime && ./prime`

Sample Output

A sample run should look like the following:

Sample 1

```
Enter an integer: 2100
All of the prime factors of 2100 are: 7 5 5 3 2 2
```

Sample 2

```
Enter an integer: 17
All of the prime factors of 17 are: 17
```

In case that the stack is full before all prime factors are found, the output should say
`Some of the prime factors of number are: ...`

Working with an array-based Queue

For the second task, we will work with an array-based queue. A queue is a similar data structure to the stack. A general queue interface is shown to you in Figure 2.

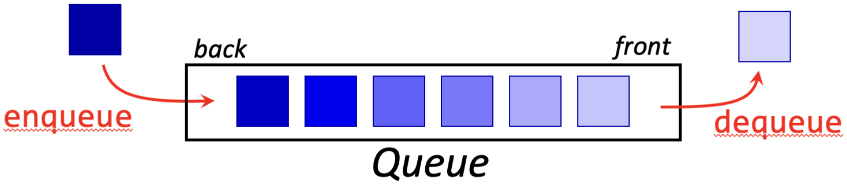

Figure 2: ADT Queue

The core methods for the queue are `isEmpty()`, `enqueue()`, `dequeue()`, and `peekFront()`. The started code includes implementations of each of the four methods mentioned above. Note how the `front` and `back` positions are very important for the queue implementation. For a queue, you always add at the *back*, and remove from the *front*.

Here is a quick recap of circular arrays and how they are used for the Array-based Queue implementation. If you look at Figure 3, you'll see how `front` and `back` move when dequeue or enqueue operations are performed.

What happens when back reaches `DEFAULT_CAPACITY - 1`? See how the implemented functions make use of the % operator.

Once you are more familiar with the existing code, implement these new functions in `ArrayQueue.cpp`:

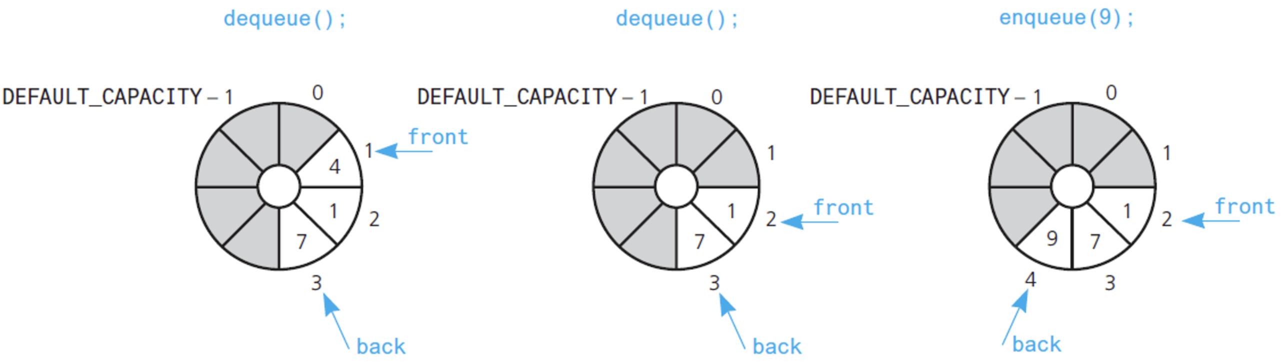

Figure 3: Array-based implementation of Queue

- `int size() const;` Return the size of the queue.
- `void clear();` Clear the queue so that it becomes empty.
- `bool inorder() const;` Returns true if the items on the queue are in order from largest (front of the queue) to smallest (back of the queue). Otherwise, returns false.

This is not a checkpoint. But we suggest at least getting familiar with the code in `ArrayQueue.cpp` while in lab. Ask us questions!

Your starter code also has a client function implemented called **`areEqual`** to help with testing your Queue code. You do not need to make any changes to it, but make sure to take a look at it as an example of working Client code.
Test your new code by compiling and running the `QueueTest.cpp` program:

```
clang++ -g -Wall QueueTest.cpp PrecondViolatedExcept.cpp -o queuetest && ./queuetest
```

Sample Output

Your output should look as follows:

```
Adding 10
Adding 9
Adding 8
Adding 7
Adding 6
Adding 5
Adding 4
Adding 3
Adding 2
Adding 1

Size method correctly returns 10.

Inorder correctly returns true on queue sorted largest (front) to smallest.

Inorder correctly returns false on out-of-order queue.

Clearing queue.
```

```
Queue is empty.

Testing inorder() for empty queue.
Correct value of true returned.

Testing areEqual for two queues that are the same.
Correct value of true returned.

Testing areEqual for two queues that are not the same.
Correct value of false returned.
```

What to Turn In

Submit your source code and your discussion log: `~csci132/bin/submit`

Be sure that the program prologue for each file you submit contains your name, course, date, and purpose of the program or a description of the contents of the file.

Reminder: Be sure to save a copy of each file in your labs directory. It is your responsibility to keep the copies of all programs written for this course until your graded assignment is returned.